Querying Tagged PDF

TEI — Text Encoding Initiative

During the last part of the last year and half of this year me and Hans were busied with some typical stuffs of the current ‘Digital Humanities’:

- bibliographic problem ($\longrightarrow$ publications module)
- critical editions in XML. Nothing published on it (standard way to map XML to ConTEXt) but we fixed a nasty bug of luajitTEX (and partially on LuaTEX) in the hash function of LUA.

TEI — Text Encoding Initiative

The XML used in these cases are very likely to be conforming to the ‘standard’ defined by TEI — the Text Encoding Initiative, a no-profit organization which publishes the Guidelines (more than 1600 pages in the PDF version). The texts are mainly in the humanities, social sciences and linguistics (verse, drama, spoken text, dictionaries, and manuscript materials.)

TEI — Text Encoding Initiative

Also

> “
>
> The TEI maintains a library of XSL stylesheets developed by Sebastian Rahtz, which can convert TEI XML files to HTML, LaTeX, or XSL:FO documents. These stylesheets are designed for specific purposes and are not intended as general-purpose conversion tools. Other XSL and CSS stylesheets are listed in the stylesheets section of the TEI wiki”

http://www.tei-c.org/Tools/

The XSLT stylesheets are conforming to the 2.0 version, so currently only the SAXON (JAVA) processor can use them.

TEI — Text Encoding Initiative

Almost at the same time the GUIT (Italian TEX User Group) decided that the next meeting (in Verona, October of this year) should be focused on TEX and Academia, so I decided to stay on the argument with the opera of W. Shakespeare *Romeo and Juliet* that is available as XML-TEI at http://www.perseus.tufts.edu/hopper/text?doc=Perseus:text:1999.03.0053.

TEI — Text Encoding Initiative

A simple fragment:

```
<?xml version="1.0" encoding="utf-8"?>
<TEI>
  <teiHeader type="text" >
:
 </teiHeader>
<text xml:lang="en">
 <body>
:
<div1 type="act" n="1" org="uniform" sample="complete">
  <head>ACT I</head><lb ed="F1" n="2" />
<div2 type="scene" n="1" org="uniform" sample="complete">
<head>SCENE I</head>
<stage type="setting">Verona. A public place.</stage>
<lb ed="F1" n="3" /><stage type="entrance">Enter SAMPSON and
GREGORY, <lb ed="F1" n="4" />of the house of Capulet, armed with
swords and bucklers.</stage>
:
 </body>
</text>
</TEI>
```

TEI — Text Encoding Initiative

ACT I

SCENE I

Verona. A public place.

Enter SAMPSON and GREGORY,
of the house of Capulet, armed with swords and bucklers.

Sam. Gregory, o’ my word, we’ll not carry coals.

Gre. No, for then we should be colliers.

Sam. I mean, an we be in choler, we’ll draw.

Gre. Ay, while you live, draw your neck out o’ the collar.

TEI — Text Encoding Initiative

There is nothing special to do to typeset this XML:

```
\startxmlsetups xml:tei:Perseus:text:1999:03:0053-setups
      \xmlsetsetup{#1}{*}{xml:tei:*}
\stopxmlsetups
\xmlregistersetup{xml:tei:Perseus:text:1999:03:0053-setups}
% TEI Header
\startxmlsetups xml:tei:teiHeader
  \xmlfunction{#1}{tei:teiHeader}
\stopxmlsetups
\startxmlsetups xml:tei:fileDesc
  \xmlfunction{#1}{tei:fileDesc}
\stopxmlsetups
```

TEI — Text Encoding Initiative

Basically every element is mapped to a LUA function:

```
xml_func["tei:TEI"]  = TEI
xml_func["tei:teiHeader"] = header
xml_func["tei:fileDesc"]  = filedesc
xml_func["tei:titleStmt"] = titlestmt
xml_func["tei:title"]     = title
xml_func["tei:author"]    = author
xml_func["tei:editor"]    = editor
xml_func["tei:sponsor"]   = sponsor
xml_func["tei:principal"] = principal
xml_func["tei:respStmt"]  = respStmt
xml_func["tei:resp"]      = resp
xml_func["tei:name"]      = name
xml_func["tei:funder"]    = funder
```

TEI — Text Encoding Initiative

Here are some functions:

```
local function TEI(t)
   local att = {['elementname']='TEI'}
   context.startelement({"document"},att)
   context.setupelementuserproperties({'document'},att)
   lxml.flush(t)
   context.stopelement()
end

local function header(t)
   local att = {['elementname']='teiHeader'}
   att.type=t.at.type
   context.startelement({"metadata"},att)
   context.setupelementuserproperties({'metadata'}, att)
   context([[{\sc]])
   lxml.flush(t)
   context([[}]])
   context.blank()
   context.stopelement()
end
```

Tagged PDF

Initially the idea was to show how to produce a PDF/A-1a (fidelity of the visual appearance *and* preservation of the logical structure), i.e. a PDF suitable for digital archiving. Of course it’s a trivial task in ConTEXt-MKIV — just a matter to select the right backend, but also not a ‘smart’ move: the current standard is PDF/A-3 and I don’t like to present ConTEXt as an out-of-date tool. So, starting from here, I have chosen a different direction.

Tagged PDF

The PDF guide (for vers. 1.7) at Chapter 10 describes how is possible to insert information of the *logical* structure of the source document. There are several possibilities.

Marked Content: a few PDF instructions to mark a stream (or a part of it). Often used as leaf in a hierarchical tree structure.

Example:

tag properties `BDC`...`EMC`

```
/sectiontitle <</MCID 0>>BDC
BT
/F1 17.21541 Tf 1 0 0 1 353.324 528.9996 Tm
[<002F0060001C004B001C>30<006900420062006800
540032006000620051004D>25<00A4>]TJ
ET
EMC
```

Tagged PDF

Logical Structure: a set of PDF instructions to *embed* the logical structure of the source document. Example:

```
19 0 obj
<< /Type /StructElem /K 18 0 R /S /document /P 17 0 R /A << /P [ << /N
(elementname) /V (TEI) >> ] /O /UserProperties >> /Pg 15 0 R >>
endobj
```

The name of the element (`/document` in this case) is user-defined. Basically it's a way to encode a tree structure where the leaves are the Marked Content.

Tagged PDF

Finally:

Tagged PDF: a logical structure where the name of the elements are fixed. It's the base for the PDF/A-{1,2,3} ISO standards.

ConTEXt has its own set of elements (a.k.a tags) and it uses a `RoleMap` dictionary to map its tags to the Adobe one:

```
13 0 obj
<< /Type /StructTreeRoot /ParentTree 14 0 R /K 15956 0 R /RoleMap
<< /mid /Span /sectioncontent /Div /p /P /delimited /Quote /item /LI
/description /Div /document /Div /mo /Span /sectiontitle /H /math /Div
/section /Sect /label /Span /metavariable /Span /itemgroup /L /division /Div
/mrow /Span /mi /Span /ignore /Span /metadata /Div /mtext /Span /line
/Code /mright /Span >> >>
endobj
```

This is main reason why a structured PDF made by ConTEXt is a Tagged PDF.

Tagged PDF

Problem:

These tag names share nothing with TEI. We can embed the structure — but can we also save the name and the attributes of an XML element ?
The answer is positive: a structure element has a `/UserProperties` dictionary that can be used to store (key,value) pairs. I choose to use (`elementname`,<element_name>
to encode the name of the element, and (<attr_name>,<attr_valu
for attributes.
I have made a patched version of ConTEXt to support it and at the end of July Hans has made a new beta that supports `/UserProperties`.

Tagged PDF

```
local function TEI(t)
   local att = {['elementname']='TEI'}
   context.startelement({"document"},att)
   context.setupelementuserproperties({'document'},att)
   lxml.flush(t)
   context.stopelement()
end

local function header(t)
   local att = {['elementname']='teiHeader'}
   att.type=t.at.type
   context.startelement({"metadata"},att)
   context.setupelementuserproperties({'metadata'}, att)
   context([[{\sc]])
   lxml.flush(t)
   context([[}]])
   context.blank()
   context.stopelement()
end
```

Tagged PDF

```
19 0 obj
<< /Type /StructElem /K 18 0 R /S /document /P 17 0 R /A << /P [ << /N
(elementname) /V (TEI) >> ] /O /UserProperties >> /Pg 15 0 R >>
endobj

21 0 obj
<< /Type /StructElem /K 20 0 R /S /metadata /P 19 0 R /A << /P [ <<
/N (type) /V (text) >> << /N (elementname) /V (teiHeader) >> ] /O
/UserProperties >> /Pg 15 0 R >>
endobj
```

Querying Tagged PDF

The next step is quite simple: write a LuaTEX program that read the PDF and store the logical structure as a XML document. The are two ways:

- use the low level functions from the `epdf` module
- use the ‘high’ level functions from the new `poppler` library (always in the `epdf` module, but currently only in LuaTEX experimental).

I have used the last one.

Querying Tagged PDF

```
<TEI>
 <teiHeader type="text" > <!– status="new" –>
  <fileDesc>
   <titleStmt>
    <title>Romeo and Juliet</title>
    <author>William Shakespeare</author>
    <editor role="editor">W. G. Clark</editor>
    <editor role="editor">W. Aldis Wright</editor>
    <sponsor>Perseus Project, Tufts University
    </sponsor>
     <principal>Gregory Crane</principal>
     <respStmt>
      <resp>Prepared under the supervision of</resp>
      <name>Lisa Cerrato</name>
      <name>William Merrill</name>
      <name>Elli Mylonas</name>
      <name>David Smith</name>
     </respStmt>
  <funder n="org:DLI2">NSF, NEH: Digital Libraries Initiative, Phase
2</funder>
  <!– Revision –>
  <respStmt xml:id="CEW.ed"><name>CEW</name>
  <resp>ed.</resp></respStmt>
  </titleStmt>
      <publicationStmt>
```

Querying Tagged PDF

```
$> luajittex –luaonly query-taggedpdf.lua Perseus_text_1999.03.0053-2.6.0.pdf
```

Querying Tagged PDF

```
<TEI>
  <teiHeader type="text" >
   <fileDesc>
    <titleStmt>
     <title>   Romeo and Juliet  </title>
     <author>  William Shakespeare </author>
     <editor role="editor" > W. G. Clark </editor>
     <editor role="editor" > W. Aldis Wright </editor>
     <sponsor>  Perseus Project, Tufts University </sponsor>
     <principal>  Gregory Crane </principal>
     <respStmt>
      <resp>  Prepared under the supervision of </resp>
      <name>   Lisa Cerrato </name>
      <name>   William Merrill </name>
      <name>   Elli Mylonas </name>
      <name>   David Smith </name>
     </respStmt>
     <funder n="org:DLI2">  NSF, NEH: Digital Libraries Initiative, Phase
 2 </funder>
     <respStmt xml:id="CEW.ed" >
      <name>  CEW </name>
      <resp>  ed. </resp>
     </respStmt>
    </titleStmt>
    <publicationStmt>
```

Querying Tagged PDF

Once we have a XML document we can query it in the same manner of any other XML document.

Problem: we have to extract the *complete* document and this takes time (even if it's done only once). So the next (bigger) problem is to find the best model to minimize the query time.

Querying Tagged PDF

Other problems:

- comments, processing instructions — is it also possible to save them ?
- blanks spaces of the XML source
- refer each element to its page (done)
- refer each element to its position on the page (is it possible ?)
- usually PDF are compressed and this introduces a significant slowdown (for short documents we can use uncompressed PDF — tech. sheet, database reporting…)

That’s all !

Thank you Folks !